Database manipulation

DB2 Information Management Software

http://www-136.ibm.com/developerworks/db2

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

What is this tutorial about?

This tutorial covers the fundamental concepts of data manipulation in DB2 databases, including the following topics:

- Changing data
- Querying a database across multiple tables
- Querying tables across multiple databases (i.e., *federated* databases)
- Using DB2 SQL functions
- Using common table expressions
- Determining when to use cursors in an SQL program
- Identifying types of cursors
- Identifying the scopes of cursors
- Manipulating cursors
- Managing a unit of work (i.e., a *transaction*)

This is the second in a series of seven tutorials that you can use to help prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The material in this tutorial primarily covers the objectives in Section 2 of the exam, entitled "Data manipulation." You can view these objectives at: *http://www.ibm.com/certify/tests/obj703.shtml*.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, you can download a free trial version of *IBM DB2 Universal Database* Enterprise Edition for reference.

Who should take this tutorial?

To take the DB2 UDB V8.1 Family Application Development exam, you must have already passed the DB2 UDB V8.1 Family Fundamentals exam (Exam 700). You can use the DB2 Family Fundamentals tutorial series (see Resources on page 25) to prepare for that test. It is a very popular tutorial series that has helped many people understand the fundamentals of the DB2 family of products.

This tutorial is one of the tools that can help you prepare for Exam 703. You should also review Resources on page 25 at the end of this tutorial for more information.

Application development involves data retrieval and manipulation. In DB2, these processes include several methods. While these methods can be programmed in different languages, the concepts remain same no matter what the

implementation language. This tutorial is the first step you should take before embarking on programming applications for DB2.

About the authors

Sunil Sabat is the technical alliance manager for PeopleSoft. He has earned three degrees: an M.S. in computer engineering, an M.S. in computer science, and an M.B.A. He is an IBM Certified IT Specialist. Having worked on all major databases for more than 10 years, Sunil has a passion for database technology and its optimized use to solve business and human problems. He currently manages the DB2 and Informix technical alliance with PeopleSoft's tools technology group. You can contact Sunil at *sunil_sabat@yahoo.com*.

Bill Wilkins is a DB2 Universal Database consultant in the IBM Toronto Laboratory. He is a grizzled veteran of the IT industry, with over 26 years of experience, mostly in IBM data management. He currently supports IBM business partners in enabling their products to IBM DB2 and Content Management products. You can contact Bill at *wilkins@ca.ibm.com*.

Notices and trademarks

Section 2. Changing and accessing data

Changing data

Changing data is a key process to understand when designing a database application. It depends on several factors:

- The data model and metadata (What are the catalog data, types, limits, checks, etc. that you must deal with?)
- Business requirements (How do you need to identify and modify the data in the database?)
- Authorization and security at the user, table, and column level (Is a particular change allowed?)
- Interfaces to access the data (How do you interface with the changing data?)

Which DB2 capabilities should you use in the design of an application? System catalog data cannot be modified by the user. Catalog tables and views store metadata about the logical and physical definition of the data. The `SYSIBM` schema owns the tables, while views for these tables are owned by the `SYSCAT` schema. You can query the catalog to get useful information. In order to make appropriate choices, you need to consider both the database design and target environments for your application. For example, you can choose to enforce some business rules in your database design instead of including the logic in your application.

The capabilities you use and the extent to which you use them can vary greatly. The capabilities that you need to consider include:

- Accessing data using:
  - Embedded SQL, including embedded SQL for Java (SQLJ)
  - DB2 Call Level Interface (DB2 CLI), Open Database Connectivity (ODBC), and Java Database Connectivity (JDBC)
  - Microsoft specifications
  - Perl DBI
  - Query products

- Controlling data values using:
  - Data types (built-in or user-defined)
  - Table check constraints
  - Referential integrity constraints
  - Views using the `CHECK OPTION`
  - Application logic and variable types

- Controlling the relationship between data values using:
  - Referential integrity constraints
  - Triggers

    - Application logic

- Executing programs at the server using:
    - Stored procedures
    - User-defined functions
    - Triggers

The key advantage in transferring logic focused on data from the application to the database is that your application becomes more independent of the data. The logic surrounding your data is centralized in one place, the database. This means that you can change data or data logic once and affect all applications that depend on that data immediately.

This latter advantage is very powerful, but you must also consider that any data logic put into the database affects *all* users of the data equally. You must consider whether the rules and constraints that you wish to impose on the data apply to all users of the data or just the users of a single application.

Your application requirements may also help you decide whether to enforce rules at the database or at the application. For example, you may need to process validation errors on data entry in a specific order. In general, you should do this type of data validation in the application code. You should also consider the computing environment where the application is used. You need to consider the difference between performing logic on the client machines and running the logic on the (usually more powerful) database server machines using either stored procedures, UDFs, or a combination of both. In some cases, the correct approach is to include enforcement in both the application (perhaps due to application-specific requirements) and in the database (perhaps due to other interactive uses outside the application).

Accessing data

In a relational database, you must use SQL to access your desired data. However, you have a choice as to how to integrate that SQL into your application. You can choose from the following interfaces and their supported languages:

- Embedded SQL
- C/C++
- COBOL
- FORTRAN
- The Java language (via SQLJ or JDBC)
- REXX
- DB2 CLI and ODBC
- Microsoft specifications, including ADO, RDO, and OLE DB
- Visual Basic, Visual C++ , and .NET languages

- Perl DBI
- Perl
- Query products like Lotus Approach, IBM Query Management Facility, Microsoft Access, or Microsoft Excel

Your program must establish a connection to the target database server before it can run any executable SQL statements. This connection identifies both the authorization ID of the user who is running the program and the name of the database server on which the program is to be run. Generally, your application process can only connect to one database server at a time; this server is called the *current* server. However, your application can connect to multiple database servers within a multisite update environment. In this case, only one server can be the current server.

Your program can establish a connection to a database server either explicitly, using a connect statement, or implicitly, by connecting to the default database server. Java applications can also establish a connection through a `Connection` instance.

Querying a database across multiple tables

You can query data from one or more tables using a `SELECT` statement. You need proper authorization to access the data that you query. The data returned is known as a *result set*.

A `SELECT` statement only specifies the criteria for the data that a result set must fetch. It does not specify the manner in which DB2 returns it. The DB2 optimizer makes the latter decision by constructing an *access plan* based on current database statistics from the system catalog tables and the type of plans it has been instructed to consider.

Let's look at some sample `SELECT` statements. The following statement selects all store names and product names from the store and product tables:

```
SELECT a.store_name, b.product_name from STORE a, PRODUCT b
```

`Store_name` is a column in the table named *store*. `Product_name` is a column in the table named *product*.

Let's look at another example. In the employee table, we'll select the department number (`WORKDEPT`) and maximum departmental salary (`SALARY`) for all departments whose maximum salary is less than the average salary in all other departments:

```
SELECT WORKDEPT, MAX(SALARY)
FROM EMPLOYEE EMP_COR
GROUP BY WORKDEPT
```

```
HAVING MAX(SALARY) < (SELECT AVG(SALARY)
FROM EMPLOYEE
WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
```

Querying tables across multiple federated databases

A *federated system* is designed to make it easy to access data, regardless of where that data is stored. This is accomplished by creating *nicknames* for all data sources (tables and views) that the user might want to access.

A DB2 federated system is a distributed computing system controlled by a *federated server*. In a DB2 installation, any number of DB2 instances can be configured to function as federated servers.

The federated system also includes multiple data sources to which the federated server sends queries. Each data source consists of an instance of a relational database management system plus the database or databases that the instance supports. The data sources in a DB2 federated system can include Oracle instances along with instances of the members of the DB2 family.

The data sources are semi-autonomous. For example, the federated server can send queries to Oracle data sources at the same time that Oracle applications are accessing those same data sources. A DB2 federated system does not monopolize or restrict access to Oracle or other data sources (beyond integrity and locking constraints).

To end users and client applications, the data sources appear as a single, collective database. Behind the scenes, the users and applications are interfacing with the federated database that is within the federated server. To obtain data from the data sources, they submit queries in DB2 SQL to the federated database. DB2 then distributes the queries to the appropriate data sources. DB2 also provides access plans for optimizing the queries. In some cases, these plans call for processing the queries at the federated server rather than at the data source. Finally, DB2 collects the requested data and passes it to the users and applications.

Queries submitted from the federated server to data sources must be read-only. To write to a data source (for example, to update a data source table), users and applications must use the data source's own SQL in a special mode called *pass through*.

For example, imagine that the nickname `DEPT` represents the remote table `EUROPE.PERSON.DEPT`. You can use the command `select * from DEPT` to query information in the remote table. All the underlying metadata is stored in the federated system catalog as part of setup and configuration.

To select, insert, and update data using a nickname, the privileges held by the authorization ID of the statement must include authorization at both the nickname level and authorization at the underlying table object database level. Here's how you would grant or revoke permissions for a nickname on certain

indexes from a user named Eileen:

```
GRANT INDEX ON <nickname> TO USER Eileen
REVOKE INDEX ON <nickname> FROM USER Eileen
```

You can use nicknames to create local summary tables, as follows:

```
CREATE TABLE <tablename> LIKE <nickname> DEFINITION ONLY
```

You can create federated views using nicknames as follows:

```
CREATE FEDERATED VIEW <viewname> AS SELECT <column>..
FROM <table>, <nickname>, <view> WHERE <expression>
```

You can use the SET option like `(SET node, varchar_no_trailing_blanks, plan_hints)` to alter nicknames, as follows:

```
ALTER NICKNAME OPTIONS ( SET varchar_no_trailing_blanks 'y')
```

Pass-through sessions allow clients connect to the databases using a server's native API or SQL dialect. When using objects in a pass-through session, you use the server's full name, not a nickname.

Section 3. Functions and expressions

What are DB2 SQL functions?

A *database function* is a relationship between a set of input data values and a set of result values. There are two types of functions: *built-in* and *user-defined*.

- *Built-in* SQL functions are provided with the database manager. They provide a single result value and are identified as part of the `SYSIBM` schema. Examples of built-in SQL functions include column functions such as `AVG`, operator functions such as +, casting functions such as `DECIMAL`, and other functions, such as `SUBSTR`.

- *User-defined* functions are functions that are registered to a database in `SYSCAT.FUNCTIONS` (using the `CREATE FUNCTION` statement). User-defined functions are never part of the `SYSIBM` schema. One such set of functions is provided with the database manager in a schema called `SYSFUN`.

DB2 allows users and application developers to extend the functionality of the database system by applying their own function definitions in the database engine itself. Applications based on user-defined functions perform better than applications that retrieve rows from the database and apply those functions on the retrieved data. Extending database functions also lets the database exploit the same functions in the engine that an application uses, providing more synergy between application and database. The use of functions contributes to higher productivity for application developers because it is more object oriented. For example, you may store the price for a product in U.S. dollars, but you may want a particular application to quote the price in U.K. pounds. You can use a function to accomplish this:

```
SELECT unit_price, currency('UK',unit_price) from product where product_id = ?
```

FENCED and NOT-FENCED modes

You can create functions in C/C++, the Java language, or OLE. A function can run in `FENCED` or `NOT-FENCED` modes. You should develop a function in `FENCED` mode before migrating to `NOT-FENCED` mode. A `NOT-FENCED` process is faster, as it uses DB2 agent memory, whereas a `FENCED` process runs in its own `db2udf` process. A `FENCED` process uses shared memory to communicate with the calling agent. `FENCED` functions are stored in `sqllib/function` and unfenced functions are stored in `sqllib/unfenced`.

DB2-supplied SQL functions

Let's look at examples of a few SQL functions. Our first example selects the title and price for all books in a table. If the price for a given title is `NULL`, the price shown is 0.00.

```
SELECT title, COALESCE(price, 0.00) AS price
FROM titles;
```

Next, we'll see an example that returns a company name and the number of characters in that company's name:

```
SELECT CompanyName, LENGTH(CompanyName)
FROM CUSTOMERS
```

Now let's see how we can return the five rightmost characters of each author's first name:

```
SELECT RIGHT(au_fname, 5)
FROM AUTHORS
```

This next example, using the project table, sets the host variable `AVERAGE (decimal(5,2))` to the average staffing level (`PRSTAFF`) of projects in the department (`DEPTNO`) called D11.

```
SELECT AVG(PRSTAFF)
INTO :AVERAGE
FROM PROJECT
WHERE DEPTNO = 'D11'
```

There are many other SQL functions defined in DB2 SQL Reference manual. You can always write your own SQL function if DB2 does not provide one.

Using common table expressions

A *common table expression* is a local temporary table that can be referenced many times in an SQL statement. This temporary table only exists for the duration of the SQL statements that define it. Every time the common table is referenced, the results will be the same. A temporary table is defined during an SQL statement using the `WITH` clause. Here's the syntax:

```
WITH <Common name1> AS ( <SELECT Expression>), <Common name2>
AS (<SELECT Expression), & SELECT <column> FROM <table_name> <where_clause>
```

<table_name> is either a table in the database or a <Common name> defined by a SQL statement including a WITH clause. Here's an example:

```
WITH prod_quantity  AS
(  SELECT product_id,  SUM (quantity) as  quantity
    FROM customer_order_item
    GROUP BY product_id
),

  totals  AS
(   SELECT  -1 as product_id, SUM(quantity) AS total
)

SELECT product_id, quantity
 FROM prod_quantity
UNION
SELECT product_id, totals
FROM totals
ORDER BY 1 DESC
```

In the above example, prod_quantity is defined as a common table expression. It is used along with a common table expression called totals. The final SELECT statement selects from both common table expressions.

Let's now look at another example:

```
WITH
  PAYLEVEL AS
    (SELECT EMPNO, EDLEVEL, YEAR(HIREDATE) AS HIREYEAR,
        SALARY+BONUS+COMM AS TOTAL_PAY
        FROM EMPLOYEE
        WHERE EDLEVEL > 16),

  PAYBYED (EDUC_LEVEL, YEAR_OF_HIRE, AVG_TOTAL_PAY) AS
    (SELECT EDLEVEL, HIREYEAR, AVG(TOTAL_PAY)
         FROM PAYLEVEL
         GROUP BY EDLEVEL, HIREYEAR)

  SELECT EMPNO, EDLEVEL, YEAR_OF_HIRE, TOTAL_PAY, DECIMAL(AVG_TOTAL_PAY,7,2)
    FROM PAYLEVEL, PAYBYED
    WHERE EDLEVEL =  EDUC_LEVEL
        AND HIREYEAR=  YEAR_OF_HIRE
        AND TOTAL_PAY < AVG_TOTAL_PAY
```

This common table expression includes PAYLEVEL. This result table includes an employee number, the year that the person was hired, the total pay for that employee, and his or her education level. Only rows for employees with an education level greater than 16 are included.

The listing also includes a common table expression with the name PAYBYED (short for "pay by education"). It uses the PAYLEVEL table to determine the education level, hire year, and average pay of employees hired in the same year who have identical education levels. The columns returned by this table have been given different names (EDUC_LEVEL, for example) from the column names used in the select list.

Finally, we get to the actual query that produces the desired result. The two tables (PAYLEVEL, PAYBYED) are joined to determine those individuals who have a salary that is less than the average pay for people hired in the same year. Note that PAYBYED is based on PAYLEVEL, so PAYLEVEL is effectively accessed twice in the complete statement. Both times the same set of rows are used in evaluating the query.

After you define a common table expression, you can use it in an SQL statement as you would any other table. You can use common table expression as many times as you wish. You can even create a common table expression based on a previously created common table expression.

Section 4. When to use cursors in an SQL program

What is a cursor? When do you need one?

A *cursor* is the mechanism used to manipulate multiple-row answer sets from a DB2 query. There are two situations in which you need a cursor:

- When a query returns more than one row.

  A `SELECT` statement with an `INTO` clause is very simple to code, but allows only one row to be returned. There is no clause in a `SELECT` statement that allows multiple rows to be processed directly, so a cursor must be used.

- When you want to update or delete one or more rows, but need to examine their contents first.

  The simplest way to update or delete rows is to use a statement like this:

  ```
  UPDATE staff SET salary = salary * 1.10 WHERE id = 100
  ```

  or `DELETE FROM staff WHERE id = 100`.

  However, these statements, called *searched* updates or deletes, don't allow the program to check the contents of the rows before updating or deleting. You could instead use a cursor in conjunction with a *positioned* update or delete. The latter are also known as `Update Where Current Of` and `Delete Where Current Of`, where *Current Of* refers to the row at which the cursor is currently positioned.

To use a cursor, you must declare it, open it, fetch rows from it one at a time, (optionally) update or delete the rows at which it is positioned, and close it. We'll see more details and examples in this and later sections.

How cursor support varies by DB2 interface

Cursor support and terminology varies among the various DB2 programming interfaces. Let's take a brief look at the differences. Later, in Manipulating cursors on page 19, we'll look at some examples for embedded SQL.

Basic support for cursors is provided through the SQL language itself, via the `DECLARE CURSOR`, `OPEN`, `FETCH`, and `CLOSE` statements.

You can perform positioned updates and deletes through the following syntax:

```
Update [or Delete] ... Where Current Of <cursor name>
```

Various interfaces build on SQL's cursor support in various ways. Traditional programming languages such as C, C++, and COBOL provide explicit support for the declaration and use of cursors in static and dynamic embedded SQL. Rows can only be processed one at a time in a forward direction.

The SQL Procedure Language supports cursors much as C does, but with the `WITH RETURN` clause added to support returning a result set to the caller of a stored procedure.

In the DB2 Call Level Interface (CLI), cursors are not explicitly declared, but CLI creates them automatically when the `SQLExecute()` or `SQLExecDirect()` functions are invoked. CLI provides additional capabilities that build on cursor support, such as the ability to scroll backwards, to receive an array of rows at one time, and to move forward by more than one row.

In JDBC, a cursor is created automatically when a `ResultSet` object is created. Additional capability is available that is similar to that in CLI.

SQLJ's cursor support is a essentially a blend of what's in JDBC and SQL, but in SQLJ the equivalent of a cursor is called an *iterator*.

Interactive tools such as the DB2 Command Line Processor (CLP) and Control Center do not let you use cursors directly. However, the tools themselves use cursors. When you execute a `SELECT` statement through CLP, or do a sample contents request for a table in the Control Center, a cursor is used to return the rows.

A simple example: Cursor usage in static embedded SQL

Before we discuss cursors further, let's look at an example of a very simple cursor in static embedded SQL (a C program).

```
EXEC SQL DECLARE c0 CURSOR FOR SELECT deptnumb, deptname FROM org;

EXEC SQL OPEN c0;

EXEC SQL FETCH c0 INTO :deptnumb, :deptname;
while (sqlca.sqlcode != 100)  /* continue until the end of the result set */
{
printf("    %8d %-14s\n", deptnumb, deptname);
EXEC SQL FETCH c0 INTO :deptnumb, :deptname;
}

EXEC SQL CLOSE c0;
```

This code prints the number and name of every department in the org table. The `DECLARE CURSOR` statement provides the query to be used, and the `OPEN` statement prepares the result set for the query. A `FETCH` statement is used repeatedly to move the values of the result set columns into program variables,

one row at a time, until the end of the result set is reached (`SQLCODE = +100`), at which point the cursor is closed.

Section 5. Identifying types of cursors

Cursor characteristics

There are three primary characteristics of cursors:

- The cursor *type*: read-only, updatable, or ambiguous.
- The cursor *direction(s):* forward-only or scrollable.
- The cursor *scope*.

We'll discuss these characteristics in the next few panels.

Cursor type

DB2 handles each of the three cursor types somewhat differently, with the differences being primarily in the realm of performance. Let's look at each of them.

Read-only cursors

When DB2 knows that a cursor is read-only, certain performance advantages can apply:

- DB2 is usually able to perform record blocking to retrieve multiple rows from the server at one time, and does not need to worry about acquiring locks that allow rows to be updated.
- DB2 can sometimes choose a better access plan for the query.

If you know that a cursor will not be used for updating or deleting rows, you should designate it as read-only by adding `FOR READ ONLY` (or `FOR FETCH ONLY`) to the `SELECT` statement for the cursor. A cursor will also be (automatically) classified as read-only if its `SELECT` statement is a join of multiple tables or includes such clauses as `ORDER BY` or `GROUP BY`.

Updatable cursors

A cursor is updatable if the `FOR UPDATE` clause is specified in its `SELECT` statement, meaning that rows will be updated via an `Update Where Current Of` statement. There can only be one table (or view) referenced in the `SELECT` statement. Because it must maintain data integrity, DB2 can only perform a minimal amount of optimization for updatable cursors.

The term *deletable cursor* is used in the SQL Reference as a way to help define updatable cursors; the two terms mean almost the same thing. See the description of `DECLARE CURSOR` in the SQL Reference for details.

Ambiguous cursors

As the name suggests, a cursor is *ambiguous* when DB2 cannot determine from the cursor definition whether it is read-only or updatable -- in other words, when the cursor's `SELECT` statement has neither `FOR READ ONLY` nor `FOR UPDATE` specified. For an ambiguous cursor, DB2 chooses whether to do record blocking for the select based on the value of the `BLOCKING` option on the `BIND` command for the application. If blocking is performed but updates occur, there is a negative performance impact, so it's always best to avoid ambiguous cursors when possible.

Cursor direction

The cursor support for DB2 embedded SQL applications allows only one row to be processed at a time, and only in a forward direction. In other words, each `FETCH` statement returns the next row of the result set to the application, and there is no other way for the application to obtain rows via the cursor.

CLI and the Java platform support *scrollable cursors*. These cursors can be positioned at an absolute row number in the result set (either forward or backward from the current position) or moved a relative number of rows (forward or backward) from the current position. For more information on scrollable cursors, see the fourth and fifth tutorials in this series (on ODBC/CLI and the Java platform, respectively, see Resources on page25).

Cursor scope

When we talk about the cursor *scope*, we mean the period during which it is available for fetching rows. That period begins when the cursor's `OPEN` statement has completed successfully. By default, the scope of the cursor ends when the cursor is closed or when a commit is executed. As we'll see on the next panel, having a commit end the scope can be a nuisance, but there's a way around it.

WITH HOLD cursors

Usually, applications should be written to have `COMMIT` statements executed fairly frequently. These commits cause locks to be released and they minimize concurrency issues between applications. Committing can cause a problem when it comes to cursors, however. This is where `WITH HOLD` comes in.

As an example, consider an application in which an SQL statement reads 10,000 rows using a cursor. The application checks each row's contents and

updates the row to set the value of a status column. Waiting to commit until all 10,000 rows have been processed could cause lock conflicts, so it commits after every 20 rows. By default, however, the commit closes the cursor, so the position in the result set would be lost and the application would have to do some special processing to continue properly from where it left off.

The solution to this problem is to change the cursor definition to include the `WITH HOLD` clause. This causes a commit to leave the cursor open and avoid releasing locks necessary to maintain the cursor's positioning. In other words, `WITH HOLD` extends the scope of a cursor beyond a commit. Here's an example of a `WITH HOLD` cursor:

```
declare C1 cursor with hold for select * from staff
```

`WITH HOLD` has no effect on what happens to a cursor during a rollback. The cursor is closed and all associated locks are released.

Section 6. Manipulating cursors

Overview of cursor processing

In this section, we'll look more closely at how cursors are used in embedded SQL applications. The basic steps, once again, are declare, open, fetch, update/delete (optional), and close.

To help understand the concept of a cursor, assume that DB2 builds a result table to hold all the rows retrieved by executing a `SELECT` statement. A cursor makes rows from the result table available to an application by identifying or pointing to a current row of this table. When a cursor is used, an application can retrieve each row sequentially from the result table until an end-of-data condition (that is, the `NOT FOUND` condition, `SQLCODE +100` or `SQLSTATE 02000`) is reached. The set of rows obtained as a result of executing the `SELECT` statement can consist of zero, one, or more rows depending on the number of rows that satisfy the search condition.

Declaring a cursor in embedded SQL

The syntax of the `DECLARE CURSOR` statement is very simple. Here's an example for static SQL:

```
declare C1 cursor for select * from staff
```

The use of the statement can be a bit confusing, however, because the statement is not executable. In other words, the statement is handled exclusively by the preparation phase of the embedded application, and when the `DECLARE` statement is reached during program execution, nothing happens. The work gets done when the cursor is opened. The only requirement is that the `DECLARE CURSOR` statement appear before the `OPEN` statement in the source file. It does not even need to be located within the same C function, for example. Every cursor name must be unique within the source file in which it is declared.

If you're using dynamic SQL, the `DECLARE CURSOR` statement is a bit different. Instead of including the syntax of the `SELECT` statement, you would use a statement name. That statement name must match the name used when preparing the related `SELECT` statement. For example:

```
EXEC SQL PREPARE stmt1 FROM :stringStmt;
EXEC SQL DECLARE c3 CURSOR FOR stmt1;
```

Opening a cursor in embedded SQL

Opening a cursor prepares the rows of the query result set for use by the program. It also positions the cursor before the first result row, though that row is not accessible by the program until a fetch is executed.

Often, the open is where the bulk of the query execution time is spent, particularly if there's an `ORDER BY` or `GROUP BY` clause in the select.

The syntax of an `OPEN` statement is very simple. To open a cursor named *c0*, you'd use the following:

```
open c0
```

Fetching a cursor in embedded SQL

Executing a fetch against a cursor causes the next row of the result set to be made available to the program, usually by placing the values of the result set columns into host variables. The *n*th column in the result set is placed in the *n*th host variable in the fetch.

For example, if a cursor `c0` is declared for `Select name, dept, id from staff`, the statement `Fetch c0 into :hv1, :hv2, :hv3` will place the value of the `name` column into `hv1`, `dept` into `hv2`, and `id` into `hv3`.

If any result set column is nullable, a second host identifier (the null indicator) should be used, and DB2 will store a negative value in that variable to represent a null value being returned. For example, changing the previous example to `Fetch c0 into :hv1, :hv2 **:hv2ind**, :hv3` would allow the program to know if an employee has a null department.

Usually a fetch is placed within a program loop that is written to continue until an `SQLCODE` of +100 is returned. At that point, all rows in the result set will have been fetched.

Updating and deleting rows with a cursor

As previously mentioned, a positioned update or delete can be done on the row at which the cursor is positioned. A fetch must have been performed previously and not returned a `SQLCODE` of +100 (or an error). Every row in the result set can be processed in this manner -- or none of them can, or any number in between. Here's an example:

```
exec sql declare cursor c0 for select name, salary from staff for update of dept;
exec sql fetch c0 into :hvname, :hvsal;
/* there might be program logic here to check the employee name and salary */
/* and only execute the update if some criteria apply                      */
exec sql update staff set dept = :newdept where current of c0;
```

This code retrieves employee information from the STAFF table and allows an employee's department to be updated. The `DECLARE CURSOR` statement provides the query, listing the `name` and `salary` columns as the columns to be retrieved, and indicates that the `dept` column may be updated in some rows. The `FETCH` statement places the employee and salary values in program variables. The `UPDATE` statement is used to change the value of the `dept` column in the previously fetched row to the value in the program variable `newdept`.

Although not shown here, program logic would normally be used to loop until the end of the result set was reached, and might have been used to only update certain rows.

Closing a cursor

Closing a cursor frees the internal storage for the cursor and makes the cursor unavailable for further use. The syntax is very simple:

```
close c0
```

By default, closing a cursor does not free the locks it holds. To do so, add the `WITH RELEASE` clause:

```
close c0 with release
```

This makes DB2 try to release all read locks. However, DB2 will keep all locks on updated rows and may need to keep some read locks for other operations or activities.

The cursor can be closed at any time when it's open -- that is, there is no need to fetch the entire result set before closing the cursor. Once a cursor is closed, it can be opened again, and it will behave as if it had not previously been used.

Section 7. Managing a unit of work

What is a transaction?

The term *unit of work,* or UOW, is synonymous with the concept of a *transaction*. It is defined as zero or more SQL queries that execute as a single atomic operation. For example, when a customer makes an online purchase from the IBM Web site, there are three steps that must be carried out:

1. The inventory of the IBM Web mall must be updated.
2. The customer must be charged for the items purchased.
3. Each item purchased must be shipped.

What would happen if the inventory records were updated and the customer was charged, but a shipping order entry was never created? Not only would you have an angry customer who never received his or her purchase, but you would introduce an inaccuracy into the inventory. Thus, all SQL queries for the purchase must be defined as a single atomic operation.

Steps in a transaction

Let's see what steps need to go into a DB2 transaction. Before you start, a connection must be established with the database against which the transaction will execute.

Start the transaction with an executable statement. An executable statement always occurs within a transaction. If a program contains an executable statement after a transaction ends, it starts a new transaction. The following are *not* executable statements:

```
BEGIN DECLARE SECTION
END DECLARE SECTION
INCLUDE SQLCA
INCLUDE SQLDA
DECLARE CURSOR
WHENEVER
```

End the transaction in the following way:

```
COMMIT
ROLLBACK
```

The `COMMIT` statement ends the transaction and makes the changes visible to other processes. Remember, you should commit regularly and do so exclusively before program termination. DB2 automatically rolls back transactions if you do

not explicitly commit them on Windows operating systems. On other operating systems, DB2 commits all pending transactions during program termination automatically .

The `ROLLBACK` statement returns the database to the state it was in before the transaction ran. The rollback prevents the changes being applied to the database after the last transaction commits. This ensures that either all operations in the transaction are committed, or none are. DB2 rolls back changes under following conditions:

- A log full condition
- A system condition that causes the system process to end

To avoid loop failure, use `WHENEVER SQLWARNING CONTINUE` or `WHENEVER SQLERROR CONTINUE` before a rollback statement. A rollback statement has no effect on contents of host variables.

You should terminate your application by taking the following steps:

- End your current transaction by `COMMIT` or `ROLLBACK`
- Release your connection by issuing a `CONNECT RESET` statement
- Free up resources (temporary storage, data structures, shared memory etc.)

You can have multiple transactions in an application. Within a transaction, you can have multiple connections to the databases.

Connections and transactions

Programming interfaces have two types of connections: *transactional* and *nontransactional.* Although DB2 supports these concepts, you should be aware that there is really only one type of connection to the database -- a transactional connection. Thus, every SQL query is part of a transaction. When you run in nontransactional mode, the programming interface you are using has enabled a feature called *autocommit* which issues a `COMMIT` statement implicitly after every SQL operation. You must ensure that you do not have autocommit enabled if your UOW has multiple queries.

Savepoints and transactions

A *savepoint* is a mechanism for undoing work performed by the DBMS when a database request fails. Savepoints make non-atomic database requests behave atomically. If an error occurs during execution, the savepoint can be used to undo changes made by the transaction between the time the savepoint was started and the time when the savepoint rollback was requested.

A savepoint allows you to group several SQL statements into a single executable block. Before the first substatement of the block is executed, a request to start a savepoint block is required. If one of the substatements ends in an error, only that substatement will be rolled back. This provides more granularity than a compound SQL statement, in which a single error causes the entire block to end in an error and rolls back the entire compound SQL statement. At the end of a savepoint block of statements, you can either release the savepoint or roll back to the savepoint.

Let's look at some SQL statements that enable you to create and control savepoints. To set a savepoint, issue a `SAVEPOINT` SQL statement. To improve the clarity of your code, you can choose a meaningful name for the savepoint. For example:

```
SAVEPOINT savepoint1 ON ROLLBACK RETAIN CURSORS
```

To release a savepoint, issue a `RELEASE SAVEPOINT` SQL statement. For example:

```
RELEASE SAVEPOINT savepoint1
```

If you do not explicitly release a savepoint with a `RELEASE SAVEPOINT` SQL statement, it is released at the end of the transaction.

To roll back to a savepoint, issue a `ROLLBACK TO SAVEPOINT SQL` statement. For example:

```
ROLLBACK TO SAVEPOINT savepoint1
```

Section 8. Conclusion

Summary

In this tutorial, you learned many concepts on data manipulation strategies. You learned how to:

- Access the data in multiple tables
- Access the data in federated systems
- Use SQL functions
- Use common tables
- Use cursors
- Program with transactions

While this tutorial focused on concepts, further tutorials in this series describe all the interfaces used to perform data manipulation.

Resources

- For more information on the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703), see *IBM DB2 Information Management -- Training and certification* (http://www.ibm.com/software/data/education/) for information on classes, certifications available and additional resources.
- As mentioned earlier, this tutorial is just one tutorial in a series of seven to help you prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The complete list of all tutorials in this series is provided below:
  1. *Database objects and Programming Methods*
  2. Data Manipulation
  3. *Embedded SQL Programming*
  4. *ODBC/CLI Programming*
  5. *Java Programming*
  6. *Advanced Programming*
  7. User-Defined Routines
- Before you take the certification exam (DB2 UDB V8.1 Application Development, Exam 703) for which this tutorial was created to help you prepare, you should have already passed the DB2 V8.1 Family Fundamentals certification exam (Exam 700). Use the *DB2 V8.1 Family Fundamentals certification prep tutorial series* to prepare for that exam. A set of six tutorials covers the following topics:
  - DB2 planning
  - DB2 security
  - Accessing DB2 UDB data

    - Working with DB2 UDB data
    - Working with DB2 UDB objects
    - Data concurrency
- Use the *DB2 V8.1 Database Administration certification prep tutorial series* to prepare for the DB2 UDB V8.1 for Linux, UNIX and Windows Database Administration certification exam (Exam 701). A set of six tutorials covers the following topics:
    - Server management
    - Data placement
    - Database access
    - Monitoring DB2 activity
    - DB2 utilities
    - Backup and recovery
- You can learn more about data manipulation from the *DB2 Information Center*. Look particularly to these sections:
    - DB2 Version 8 CLI Guide and Reference, Part 1. See especially Chapter 5: Cursors.
    - DB2 Version 8 SQL Reference, Volume 2. This covers the SQL statements related to cursors.
    - DB2 Version 8 Application Development Guide: Programming Client Applications. This covers cursors in different programming environments.
- Also check out the sample programs that come with DB2. These are shipped in `sqllib/samples`, with various subdirectories for C, JDBC, etc.
- Read the *DB2 UDB v8 Application Development Certification Guide*, by David Martineau, Steve Sanyal, Kevin Gashyna, and Mike Kyprianou (International Business Machines Corporation, 2003).
- Check out *developerWorks Subscription* for one-stop access to a comprehensive portfolio of the latest IBM software from DB2, Lotus, Rational, Tivoli, and WebSphere, allowing you to maximize ROI and lower your labor costs, leading to superior productivity.

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .